MONASH University

ECE3073 **Computer Systems**

Program Design and Analysis: Compilation

Embedded systems considerations:

- rich functionality
- small environment
- timing constraints
- power consumption

Acknowledgement

The lecture notes of Marilyn Wolf from
Computers as Components,
Principles of Embedded Computing System Design

Based on adaption from Dr Royan Ong,
Malaysian Campus

Minor modifications and additions by Clive Maynard 2020

A good reference is:
Real Time UML by Bruce Powel Douglass
published by Addison Wesley

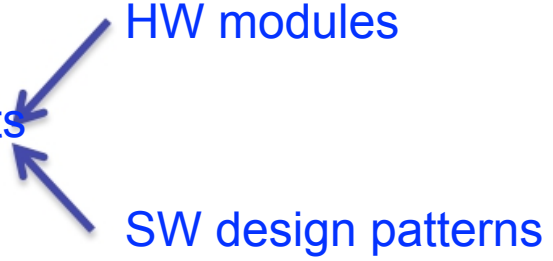
WARNING

Design patterns

- **Design pattern : generalised description of the design of a certain type of program.**

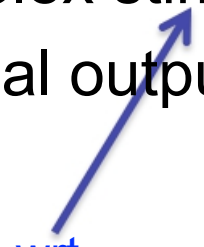
- **Designer fills in details to customise the pattern to a particular programming problem.**

Product planning is usually top down
Implementation bottom up
Implementation uses a catalogue of parts



Design Pattern Example 1 State Machine

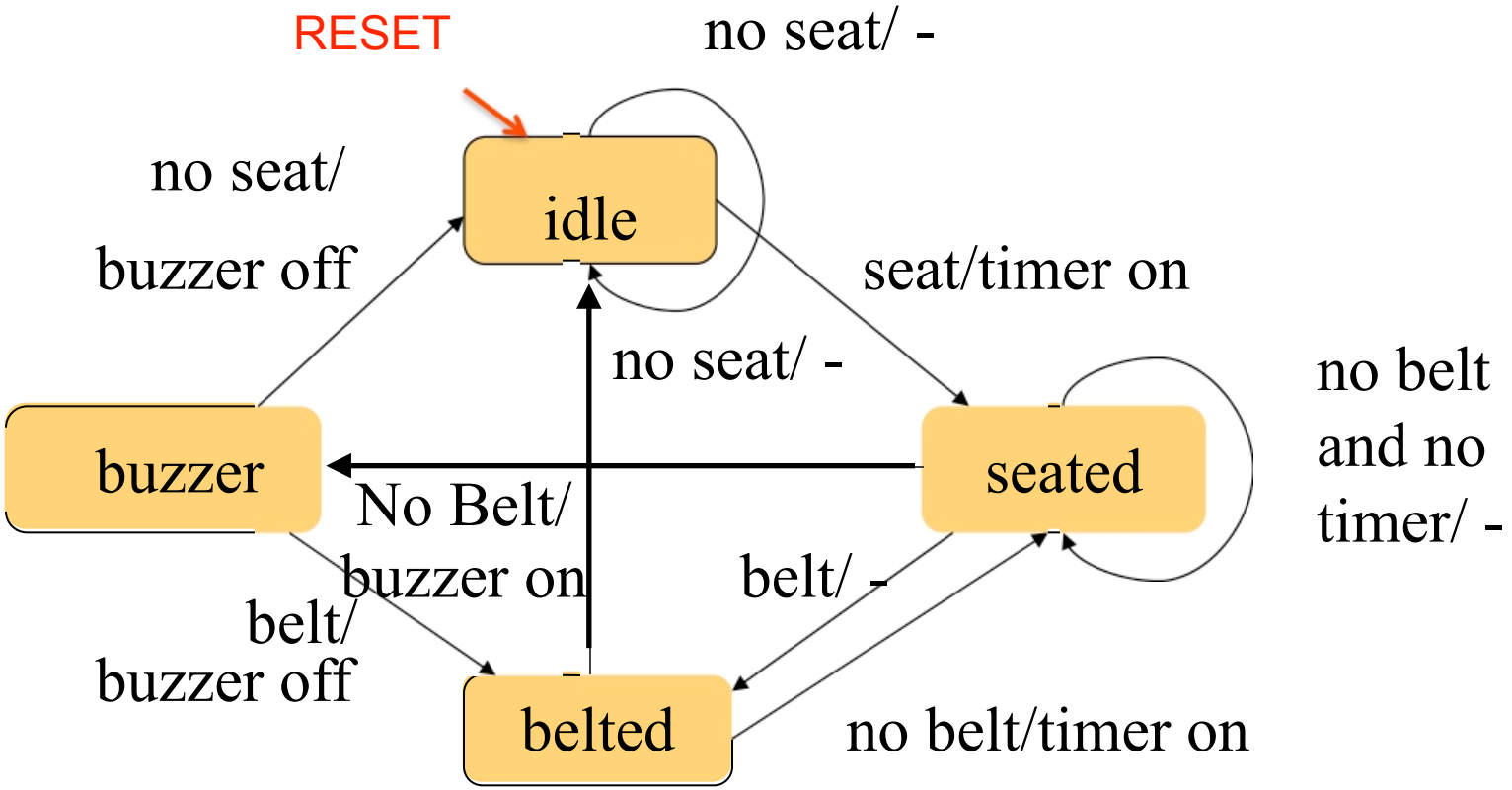
- A **State machine is useful in many contexts:**
  - parsing user input
  - responding to complex stimuli
  - controlling sequential outputs



The presentation is using a subset of UML (Unified Modeling Language). Specifically the Statecharts specifications

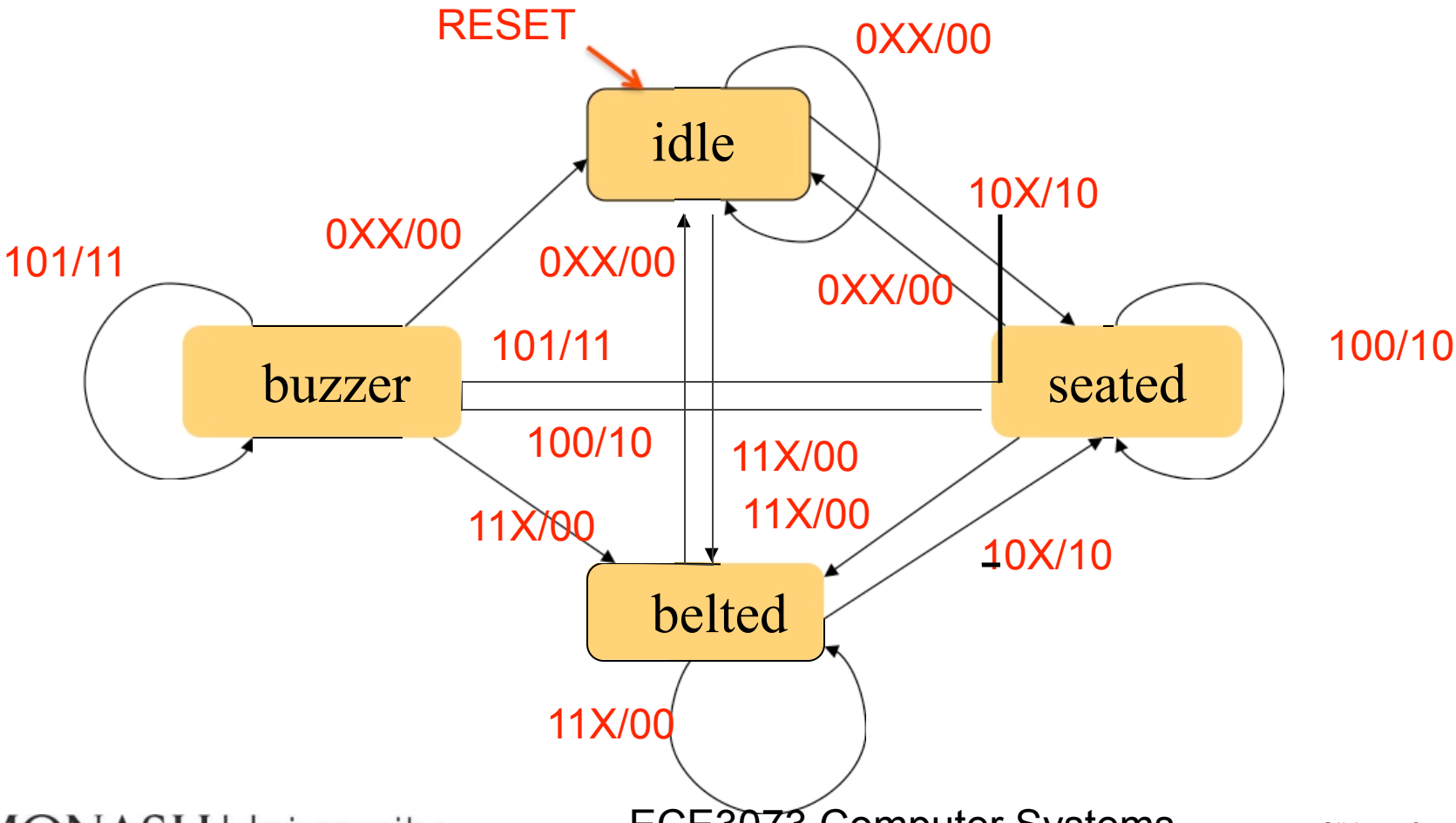
State machine example – seat belt alarm

Inputs/outputs (- = no action)



State machine example – more complete

(Seated) (Belted) (Timed_out) / (Timer_on) (Buzzer_on)



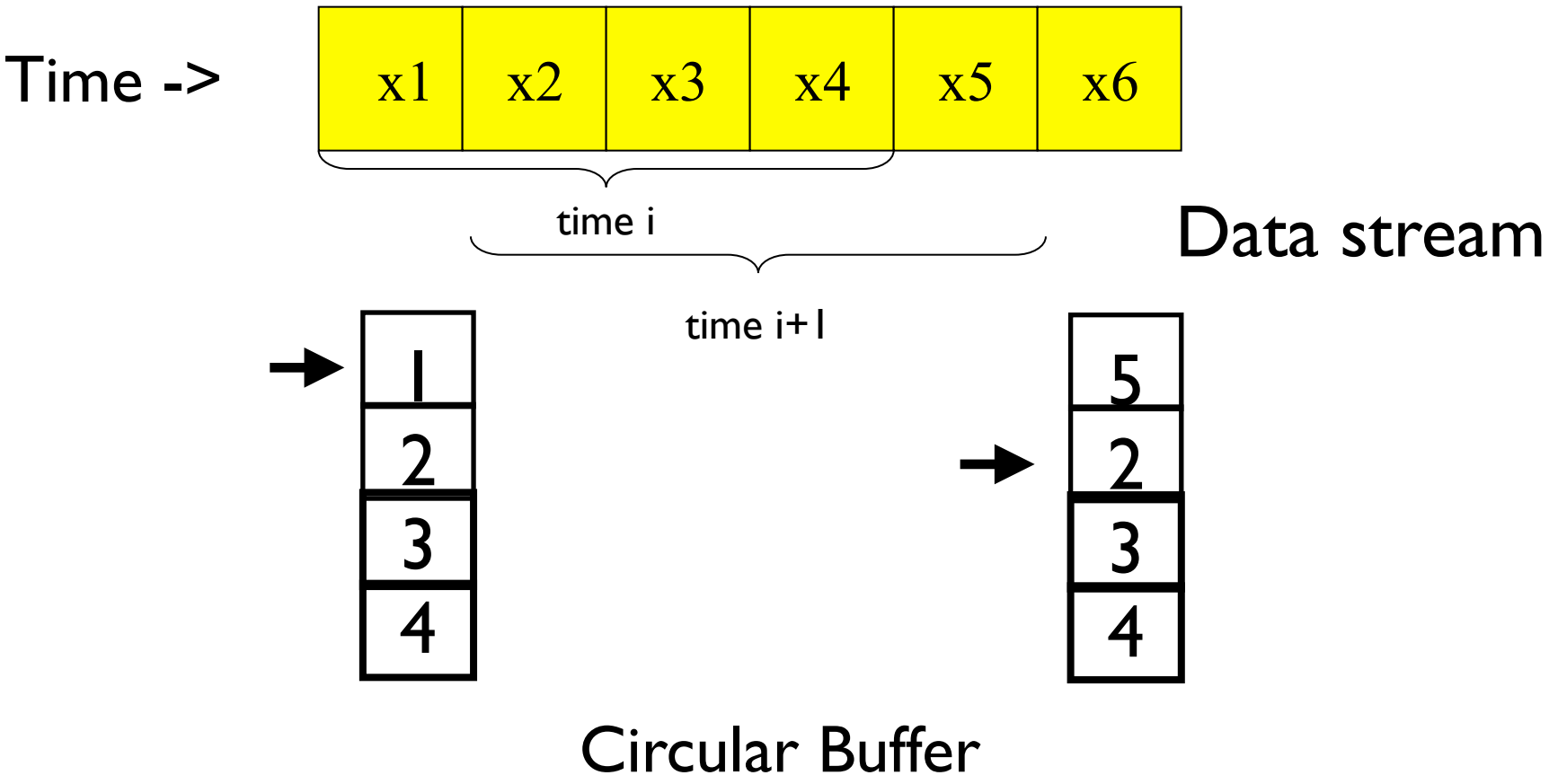
C implementation

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
    case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
        break;
    case SEATED: if (belt) state = BELTED;
                 else if (timer) state = BUZZER;
        break;
    …
}
```

const defn is better. Why?

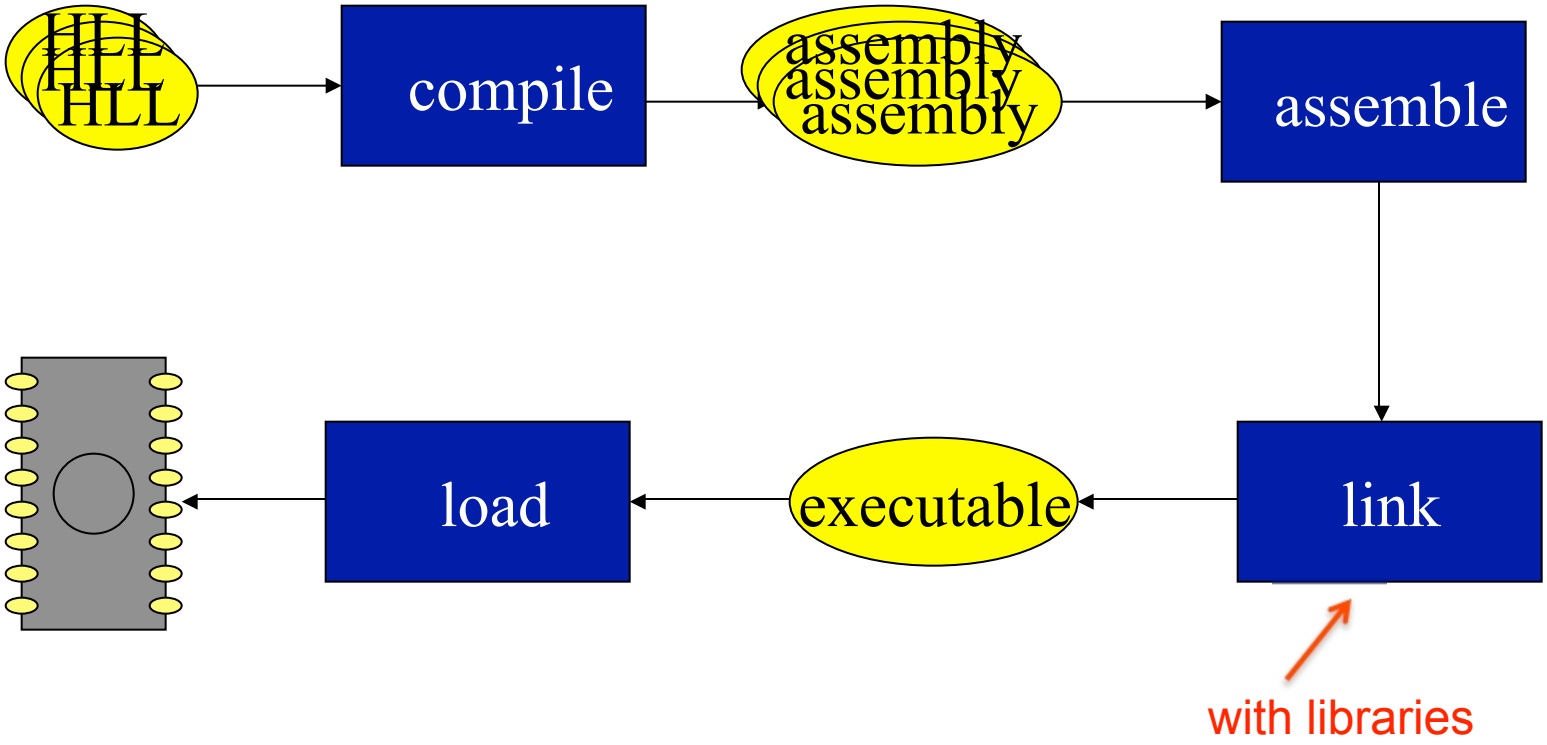
In digital logic state transitions are on clock edge. In SW what is the trigger?

Design Pattern Example 2: Circular buffer



Compilation: C to Binary

C is easier to write + can be more optimised and efficient



Models of Programs

- **Source code not a good representation for programs:**
  - clumsy
  - leaves much information implicit

- **Compilers derive intermediate representations to manipulate and optimise programs**

  Such as assembler and p- code for Pascal

- **We will use a control/data flow graph (CDFG) to model a program. As a starting point we will look at data flow graphs which deal purely with data.**

Data Flow Graph (DFG)

- **Data only does not represent control**

- **Models basic block:**
  - Code with one entry
  - Code with one exit

- **Describes the minimal ordering requirements on operations**

Data Flow Graph Definition

- **A DFG is a directed graph that shows the data dependencies between a number of functions.**
- **Nodes ‘ fire ’ when their input data is available (there may be several nodes ready to fire at a given time)**
- **Approximately, each node represents one operation that the system can perform (one assembler instruction)**

Example (1/2): DFG

Original:

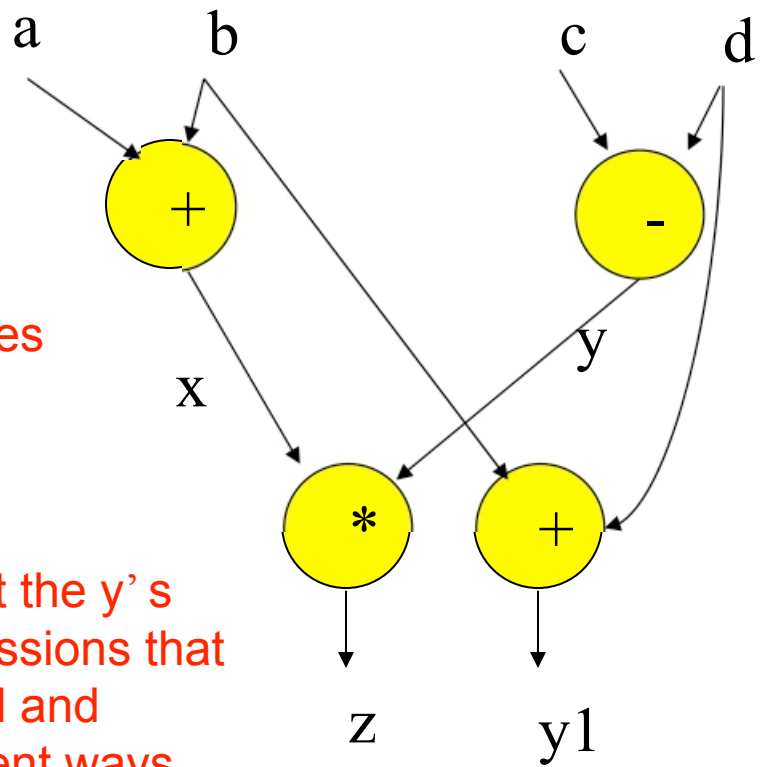
```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

To allow reordering variables appear once on LHS

Single assignment form:

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

y is overwritten but the y's are different expressions that could be optimised and reordered in different ways



Example (2/2): DFG

- **Single assignment form:**

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

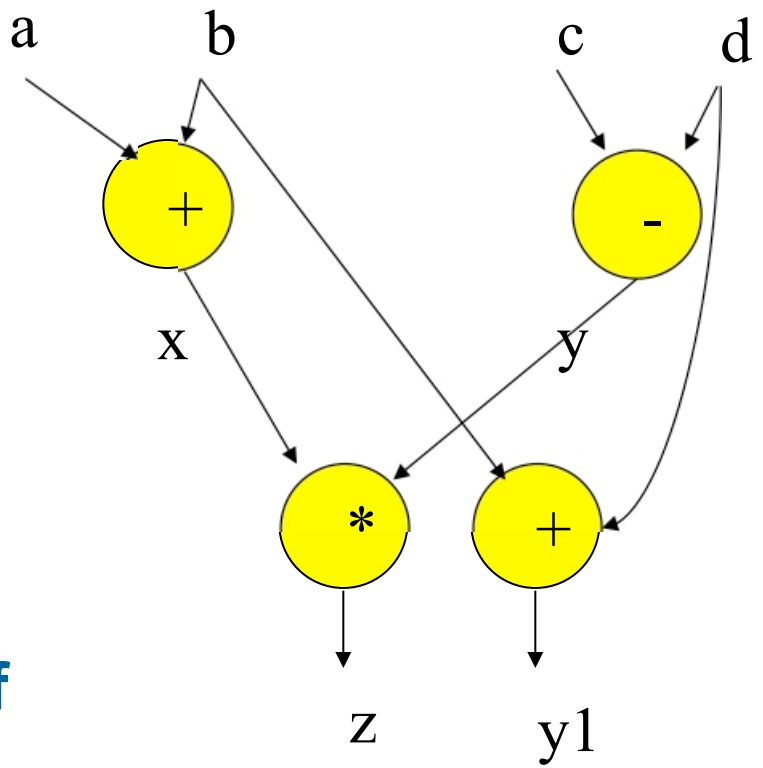
Could swap

- **Partial order:**

1) a + b, c - d
2) b + d, x * y

Must do 1 before 2

- **Can calculate each set of partial orders in any order**



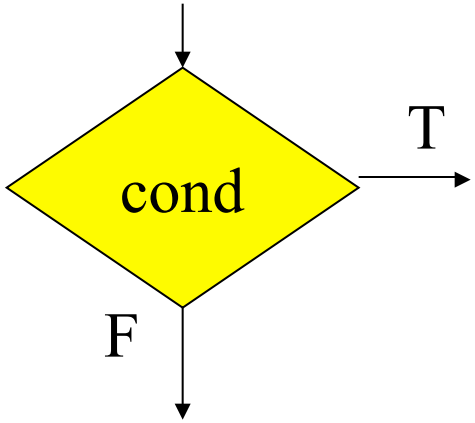
Gives opportunity for optimisation

Control- Data Flow Graph (CDFG)

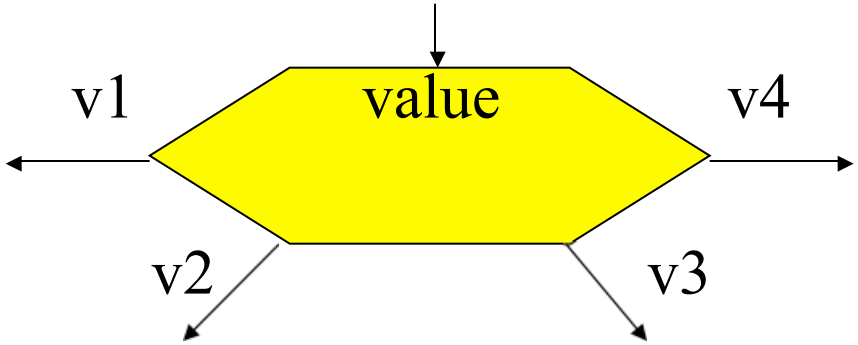
- **Represents control and data**
- **Uses DFG as components**
- **Data flow node: encapsulates DFG**
- **Used in testing each path through code**

```
x = a + b;
y = c + d
```

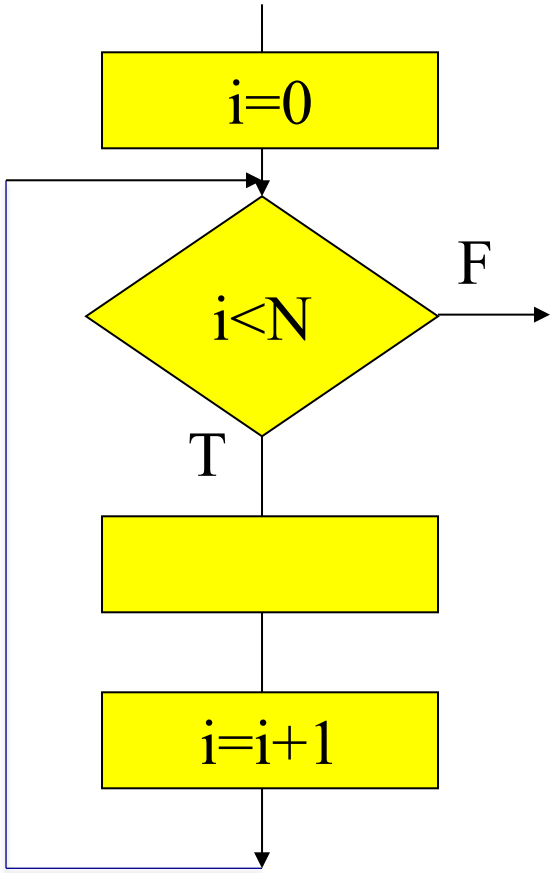
CDFG Decision Nodes (1/2)

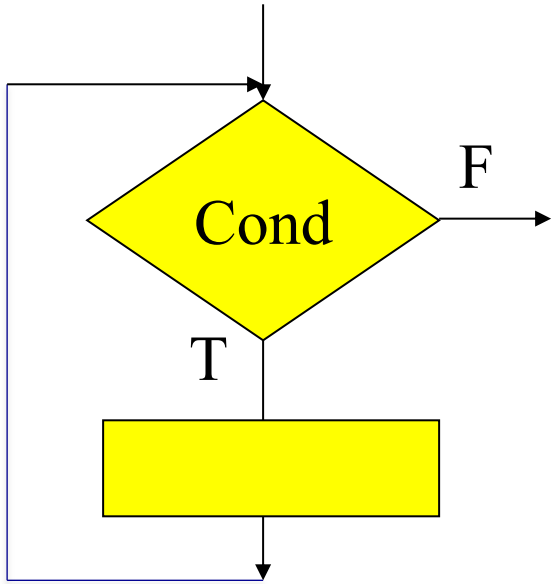


Which C statements do these correspond to?



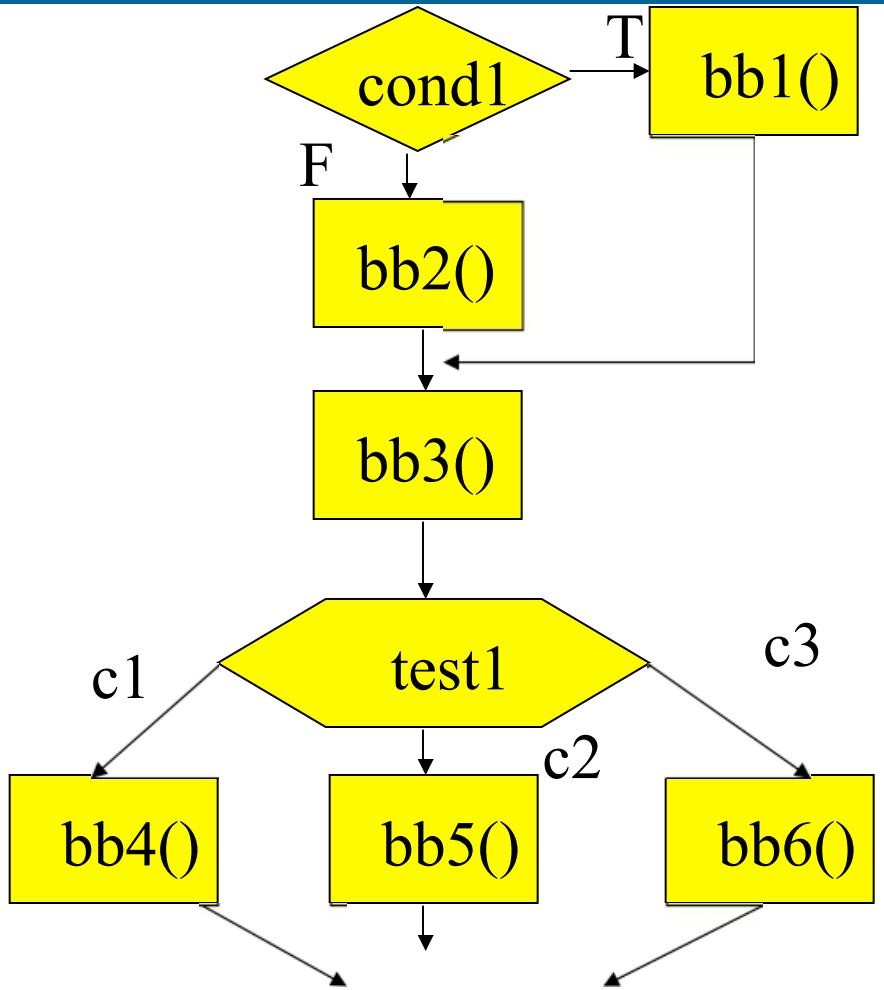
CDFG Decision Nodes (2/2)





Example: CDFG

```
if (cond1) bb1();
else bb2();

bb3();
switch (test1) {
    case c1: bb4();
                break;
     case c2: bb5();
                break;
     case c3: bb6();
                break;
     }
```

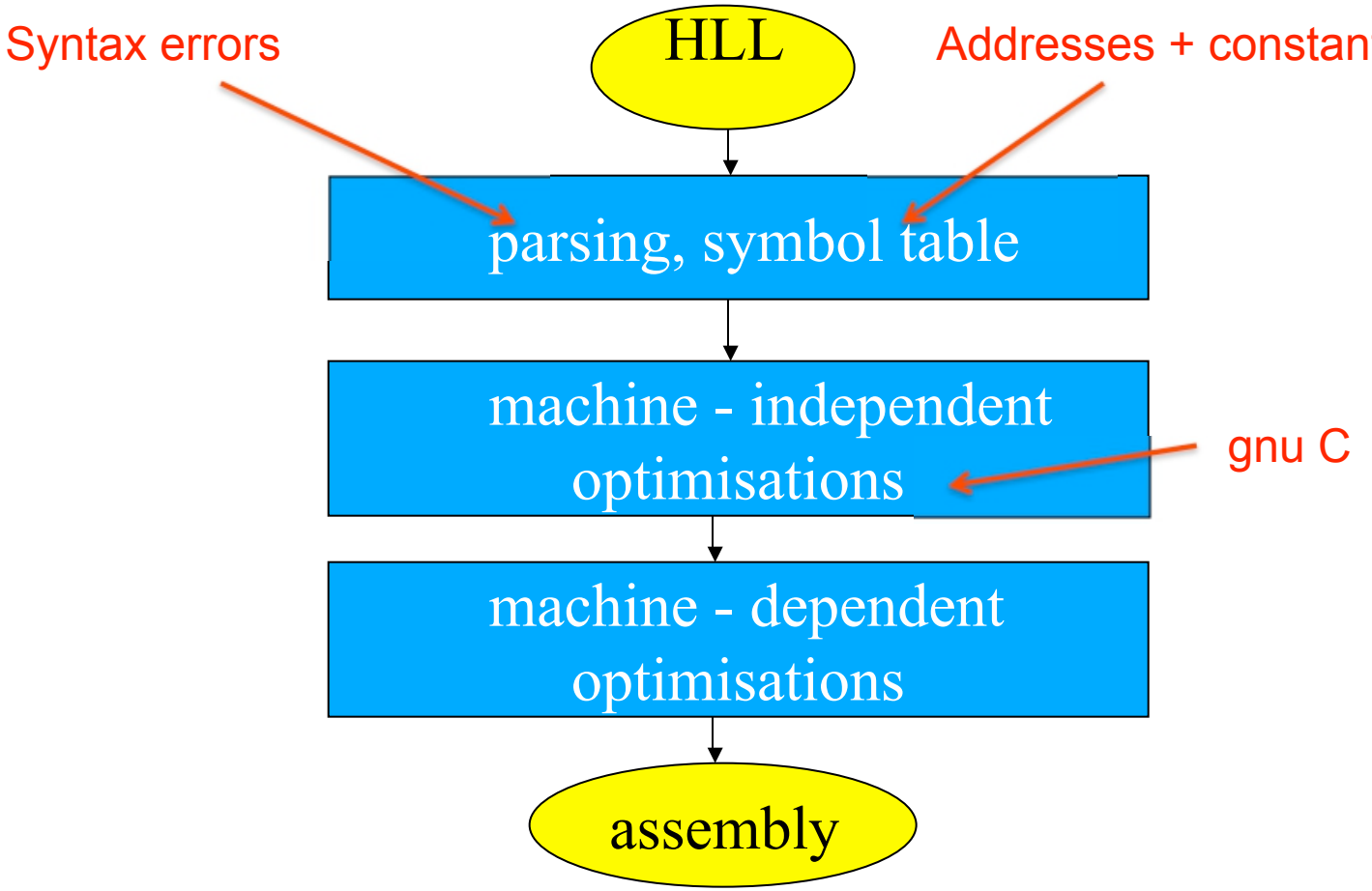


Compilation

- **Compilation strategy:**
  - compilation = translation (source code to assembly)
    + optimisation

- **Compiler determines quality of code**
  - use of CPU resources (registers, processor cores)
  - memory access scheduling
  - code size

General Compilation Phases



Source Translation and Optimisation

- **Source code translated into intermediate form such as CDFG**
- **CDFG is transformed/optimised**
- **CDFG is translated into instructions with optimisation decisions**
- **Instructions are further optimised**

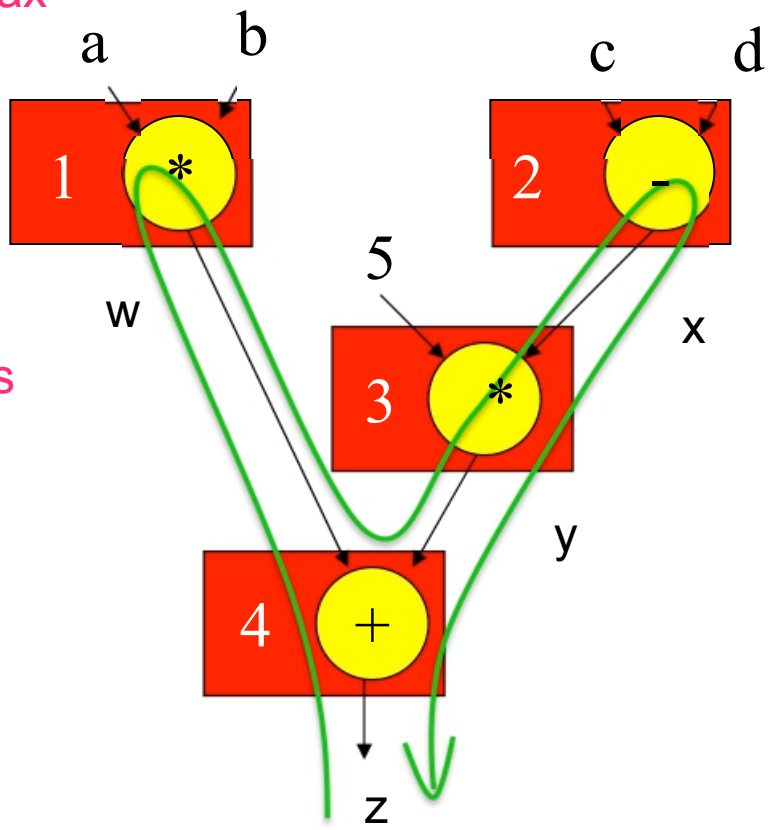
Example 1: Source to Assembly

- **Expression:**
  a * b + 5 * (c – d)

Look for syntax errors

- **Data Flow Graph**

Dependencies



Reverse Polish notation

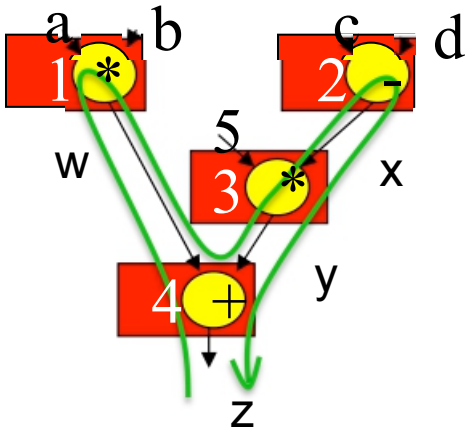
z=ab*5cd-*+

Walk the graph post order (see ECE2071)

Example 1: Source to Assembly (2/2)

w = a * b
x = c – d
y = 5 * x
z = w + y

Can reuse w, x, y



NIOS - II Code:

1	ldw	r3, - 16(fp)
	ldw	r2, - 12(fp)
	mul	r3, r3, r2
2	ldw	r4, - 8(fp)
	ldw	r2, - 4(fp)
	sub	r2, r4, r2
3	muli	r2, r2, 5
4	add	r3, r3, r2

RPL	HPcalculator
push a	a Enter
push b	b
*	*
push 5	5 Enter
push c	c Enter
push d	d
-	-
*	*
+	+

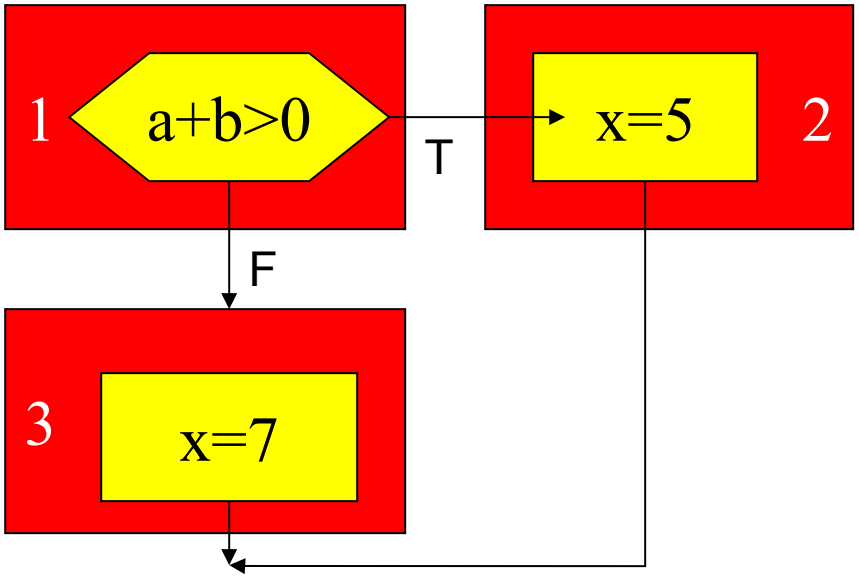
Example 2: Source to Assembly

- **Expression:**

  if (a + b > 0) x = 5;

  else x = 7;

- **Data Flow Graph:**



- **NIOS-II Code:**

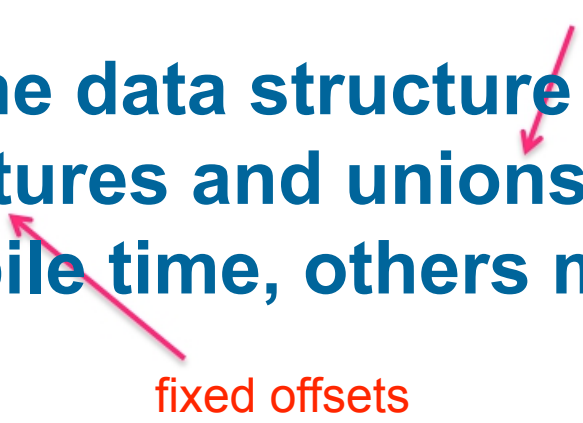
```
        a   ldw   r3,-20(fp)
        b   ldw   r2,-16(fp)
1           add   r2,r3,r2
    b       cmplti r2,r2,1      T=1, F=0
            bne r2, zero, 0xc0
            movi  r2,5
2       x   stw   r2,-4(fp)
            br    0xc8
  0xc0:     movi  r2,7
3       x   stw   r2,-4(fp)
  0xc8:
```

Compilation
Part B

Data structures

- **Different data structures use different layouts**

- **Some data structure offsets (arrays, structures and unions) can be computed at compile time, others must be computed at run time**



Exercises!

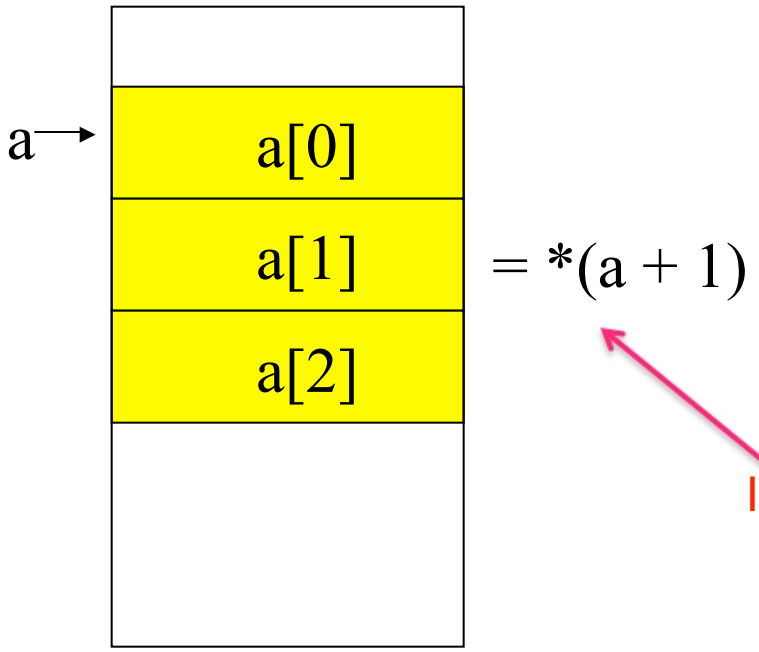
- **Check the output in assembler of the NIOS compiler for:**
  - Loops (for, while, do while)
  - Case statements
  - Data structures (structures, arrays, local, static, global)
  - Parameter passing (more parameters than registers, pointers, structures, C++ pass by reference)

This lecture will overview some aspects of these

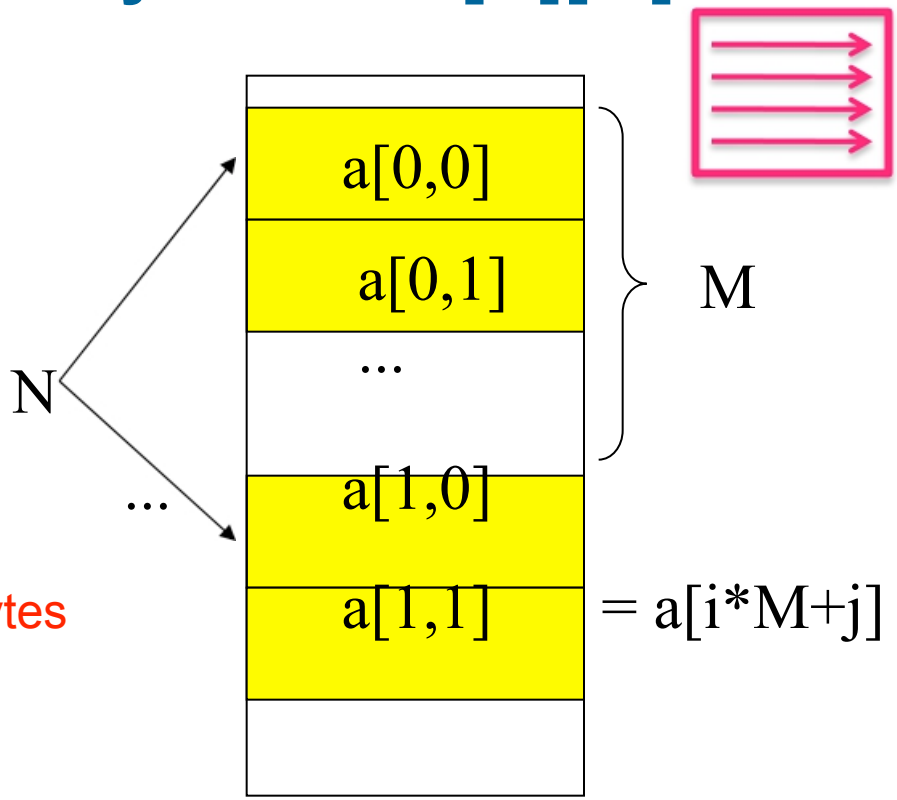
Arrays

Fortran – column major

- **1D: C array name points to 0th element**



- **2D: Row- major layout for a[N][M]**

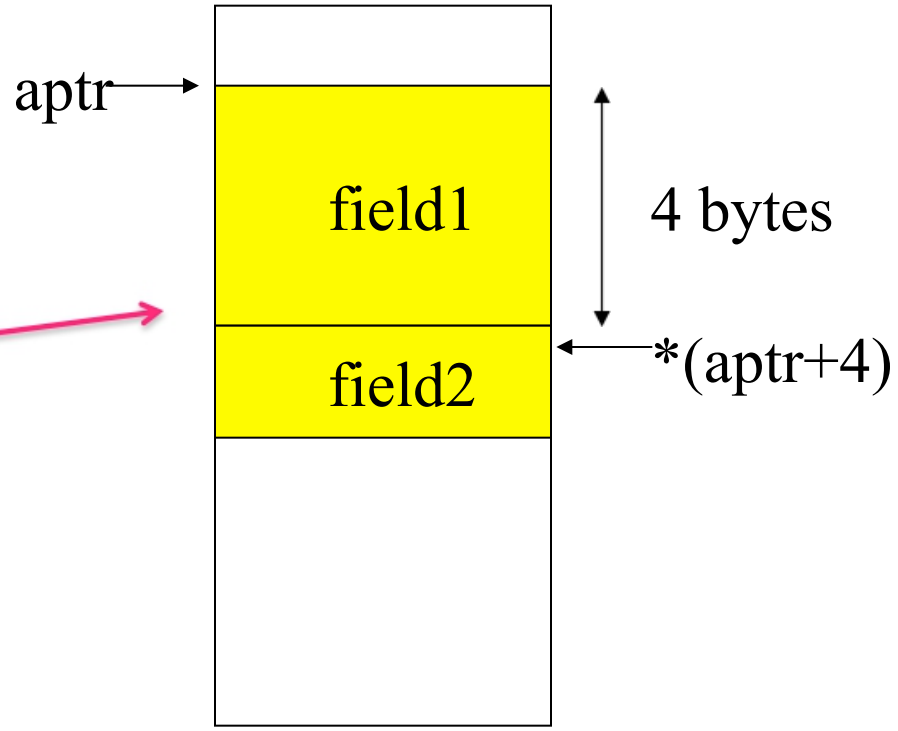


Structures

- **Fields within structures are static offsets:**

```
//structure definition
struct {
    long field1;
    char field2;
    } mystruct;

//structure declaration
mystruct a;

//structure pointer
mystruct* aptr = &a;
```

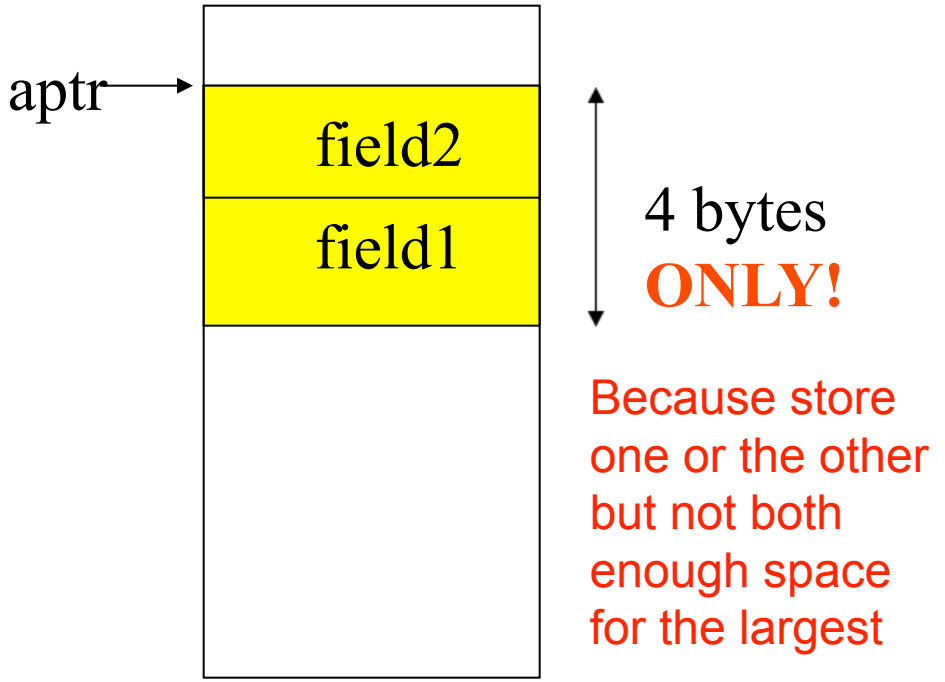
5 bytes



Unions

- **Fields within unions have same (zero) offset**

```
//structure definition
union {
    long field1;
    char field2;
    } myunion;

//structure declaration
myunion a;

//structure pointer
myunion* aptr = &a;
```



- **NOTE: field2 overlaps with field1 !!!**

Optimisation

- **Goal is to reduce:**
  - Code size
  - Execution time
  - Processor resources
- **May not be possible to achieve all goals. Usually a compromise of goals**

Expression Simplification

- **Constant folding:**
  8 + 1 = 9 `x = 8 + 1;`

  Pre- compute

  – If a result can be calculated at compile time, saves processor computation
- **Algebraic:**
  a * b + a * c = a * (b + c) `x = a * b + a * c;`

  – Addition is generally much faster than multiplication
- **Strength reduction:**
  a * 2 = a << 1 `x = a * 2;`

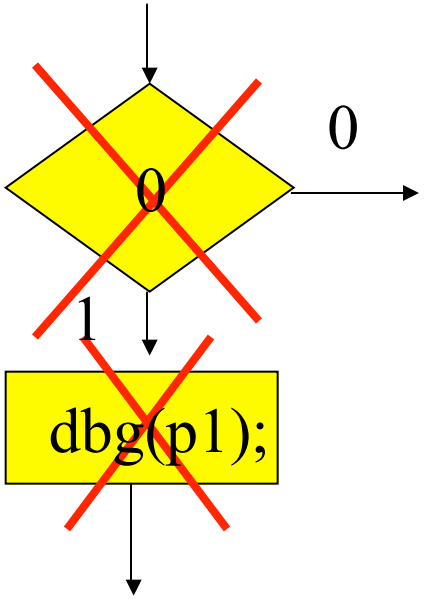
  – Left shift is multiplication by 2, right shift is integer division by 2

Dead Code Elimination

In general difficult but some cases are easy

- **Dead code:**

```
#define DEBUG 0
...
if (DEBUG) dbg(p1);
```



- **Can be eliminated by analysis of control flow, constant folding**

- **Parts of code cannot be reached! Not a good thing!**

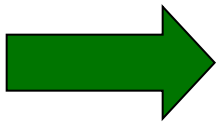
Procedure Inlining

Goes against structured programming principles

- **Eliminates procedure linkage overhead:**

```
int  foo(a,b,c)
     {
     return  a  +  b  -  c;
      }

…

z  =  foo(w,x,y);
```

But uses more memory as code is repeated each place it would have been called



```
z  =  w  +  x  -  y;
```

Compilation
Part C

Procedure Linkage

Function returning void

- **When a procedure (function) is called:**
  - May need to pass parameters to procedure
  - May need to return result from procedure
- **Example:**

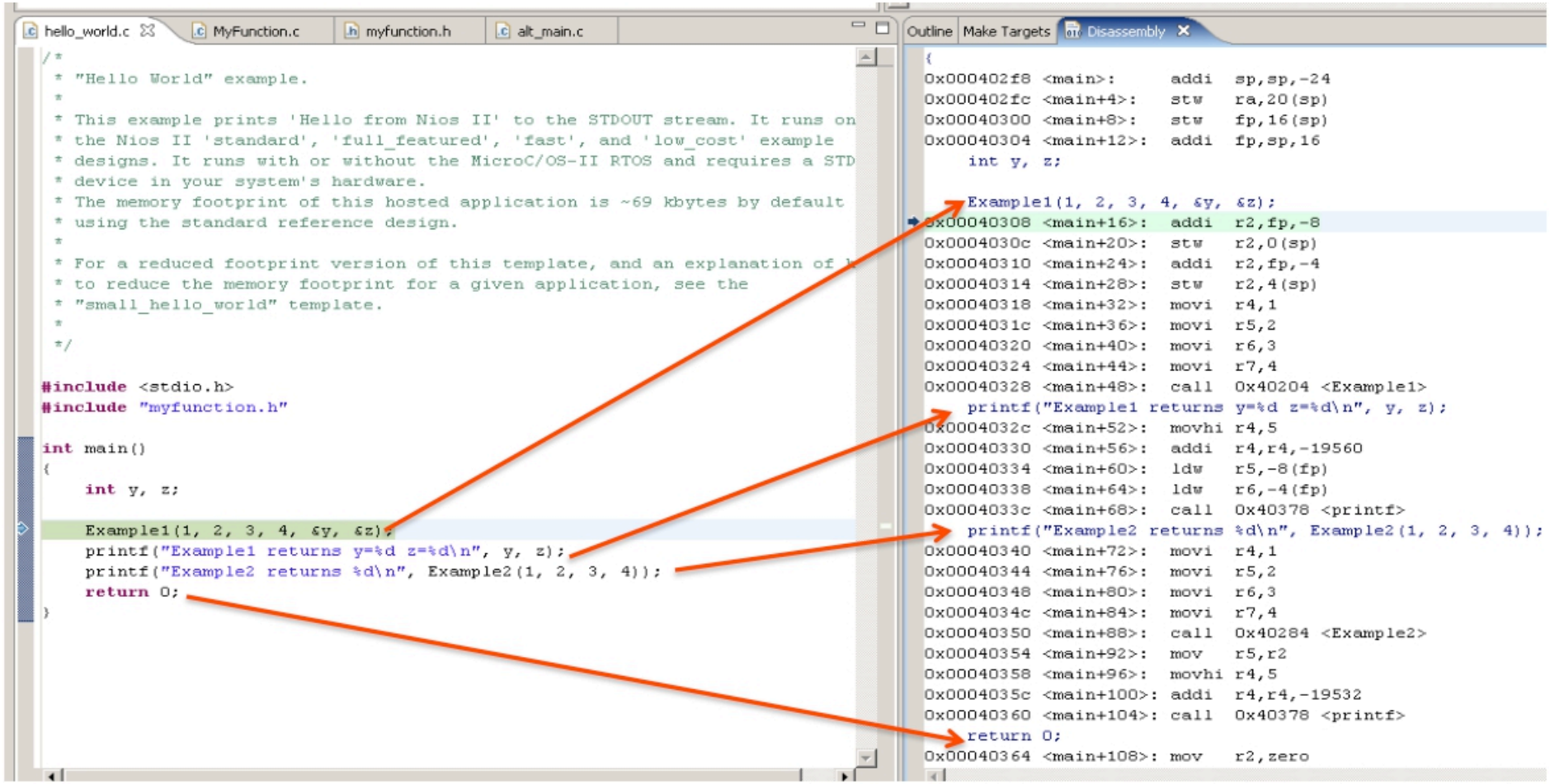
```
long Function_A (int A, int B);
void Procedure_B (char* C);
```

- **Parameters and returns passed on stack**
  - Procedures with few parameters may use registers

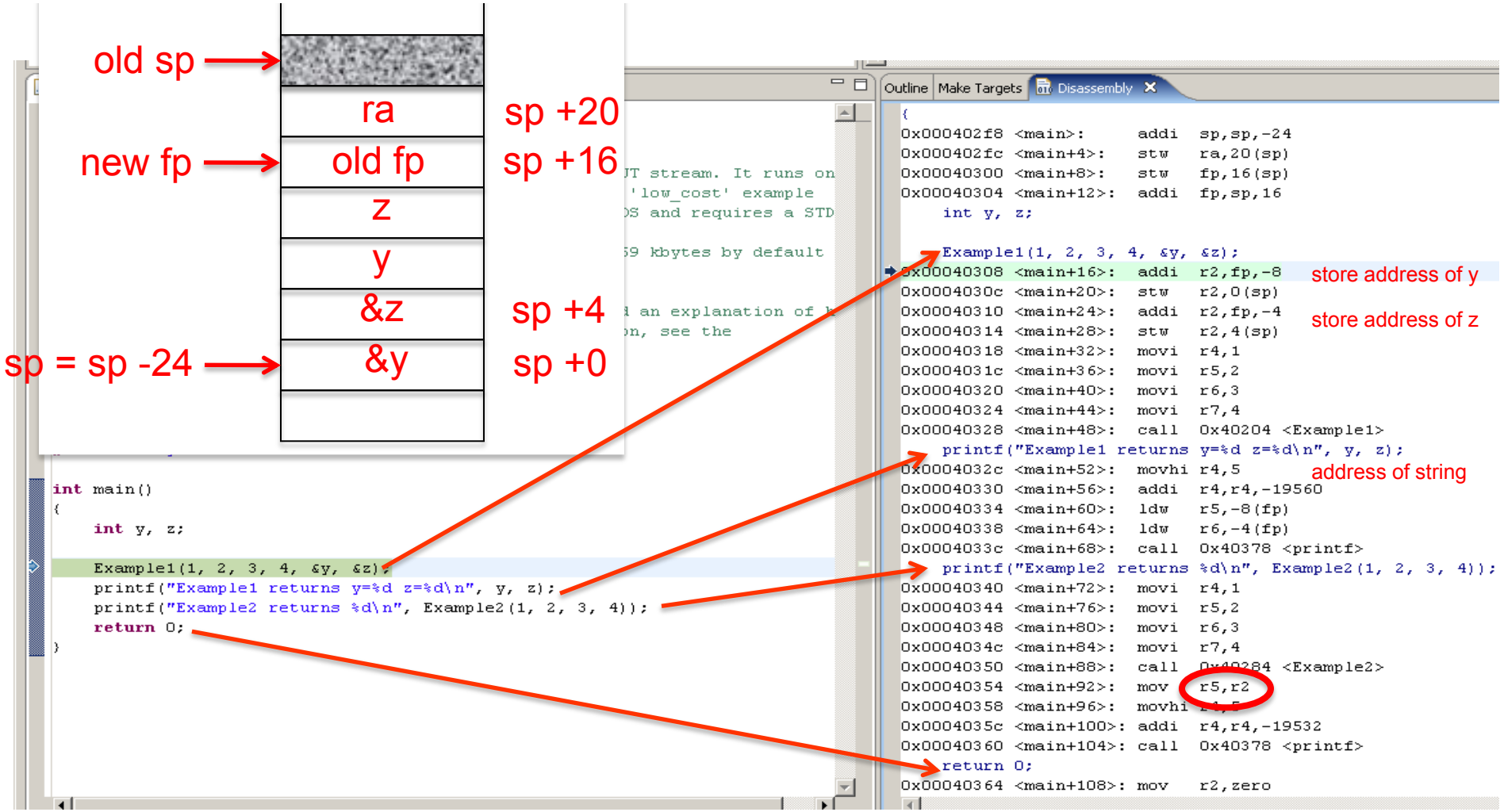
Can return results on stack or load into memory

faster

Example of Procedure Linkage – A simple 3 file project



Example of Procedure Linkage – A simple 3 file project



Common header file

hello_world.c | MyFunction.c | myfunction.h

```
#ifndef MYFUNCTION_H_
#define MYFUNCTION_H_
void Example1(int a, int b, int c, int d, int *Pz, int *Py);
int Example2(int a, int b, int c, int d);


#endif /*MYFUNCTION_H_*/
```

Prevent redefinition

Console | Tasks | Memory

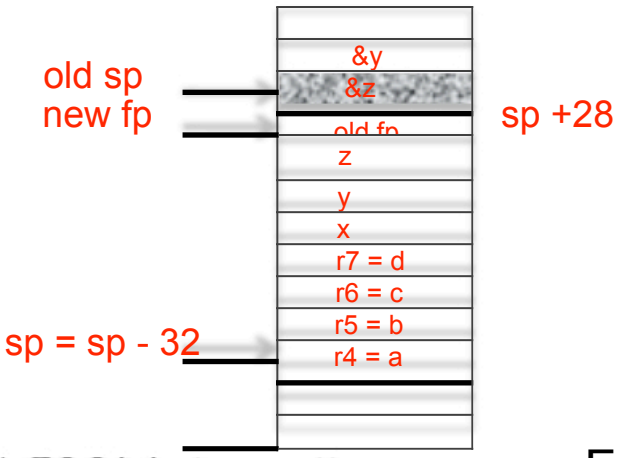
Function prototypes allow consistency type checking of parameters and return values across files

Header file is included in the user: main.c
and
the definition: MyFunction.c

Function Example1()

hello_world.c | MyFunction.c | myfunction.h

```
#include "myfunction.h"

void Example1(int a, int b, int c,
    int d, int *Pz, int *Py){
    int x, y, z;

    x = a + b;
    y = c - d;
    z = x * y;
    y = b + d;

    *Pz = z;
    *Py = y;
} /* Example1 */

int Example2(int a, int b, int c, int d){
    int x;

    if (a + b > 0) x = 5;
    else x = 7;

    return a * b + 5 * (c - d);
} /* Example2 */
```



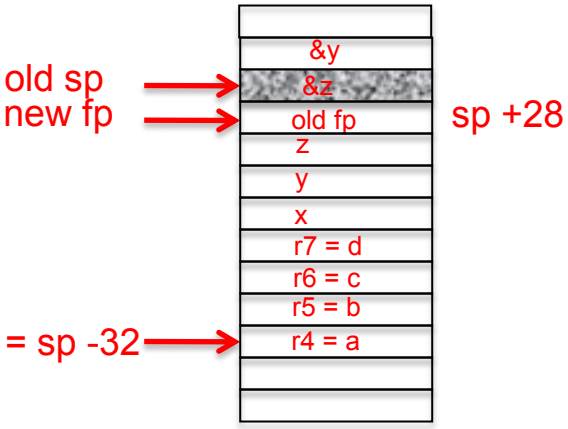
Outline | Make Targets | Disassembly

```
    int d, int *Pz, int *Py){
0x00040204 <Example1>:     addi sp,sp,-32
0x00040208 <Example1+4>:   stw  fp,28(sp)
0x0004020c <Example1+8>:   addi fp,sp,28
0x00040210 <Example1+12>:  stw  r4,-28(fp)
0x00040214 <Example1+16>:  stw  r5,-24(fp)
0x00040218 <Example1+20>:  stw  r6,-20(fp)
0x0004021c <Example1+24>:  stw  r7,-16(fp)
    int x, y, z;

    x = a + b;
0x00040220 <Example1+28>:  ldw  r3,-28(fp)
0x00040224 <Example1+32>:  ldw  r2,-24(fp)
0x00040228 <Example1+36>:  add  r2,r3,r2
0x0004022c <Example1+40>:  stw  r2,-12(fp)
    y = c - d;
0x00040230 <Example1+44>:  ldw  r3,-20(fp)
0x00040234 <Example1+48>:  ldw  r2,-16(fp)
0x00040238 <Example1+52>:  sub  r2,r3,r2
0x0004023c <Example1+56>:  stw  r2,-8(fp)
    z = x * y;
0x00040240 <Example1+60>:  ldw  r3,-12(fp)
0x00040244 <Example1+64>:  ldw  r2,-8(fp)
0x00040248 <Example1+68>:  mul  r2,r3,r2
0x0004024c <Example1+72>:  stw  r2,-4(fp)
    y = b + d;
0x00040250 <Example1+76>:  ldw  r3,-24(fp)
0x00040254 <Example1+80>:  ldw  r2,-16(fp)
0x00040258 <Example1+84>:  add  r2,r3,r2
0x0004025c <Example1+88>:  stw  r2,-8(fp)

    *Pz = z;
0x00040260 <Example1+92>:  ldw  r3,4(fp)
0x00040264 <Example1+96>:  ldw  r2,-4(fp)
0x00040268 <Example1+100>: stw  r2,0(r3)
    *Py = y;
0x0004026c <Example1+104>: ldw  r3,8(fp)
0x00040270 <Example1+108>: ldw  r2,-8(fp)
0x00040274 <Example1+112>: stw  r2,0(r3)
} /* Example1 */
0x00040278 <Example1+116>: ldw  fp,28(sp)
0x0004027c <Example1+120>: addi sp,sp,32
0x00040280 <Example1+124>: ret
```

Function Example1()

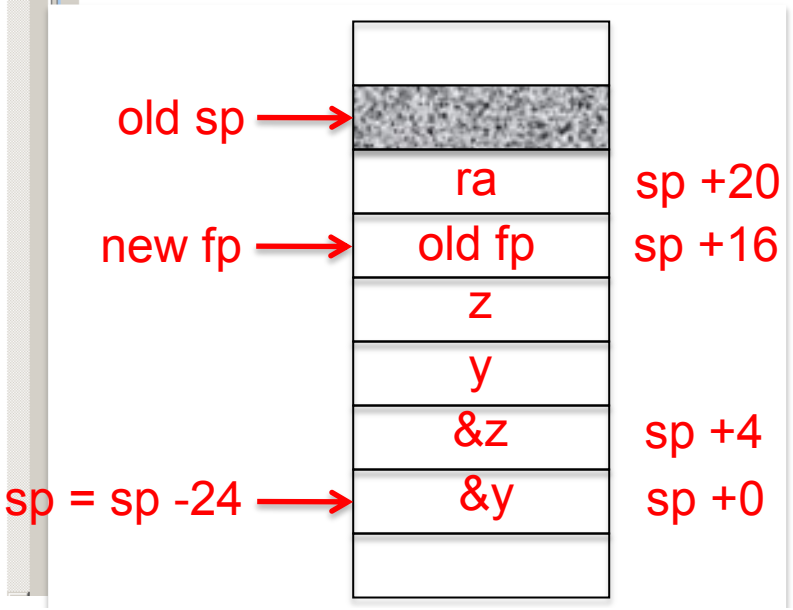
hello_world.c | MyFunction.c | myfunction.h

```
#include "myfunction.h"

void Example1(int a, int b, int c,
    int d, int *Pz, int *Py){
    int x, y, z;

    x = a + b;
    y = c - d;
    z = x * y;
    y = b + d;

    *Pz = z;
    *Py = y;
} /* Example1 */

int Example2(int a, int b, int c, int d){
    int x;

    if (a + b > 0) x = 5;
    else x = 7;

    return a * b + 5 * (c - d);
} /* Example2 */
```



Outline | Make Targets | Disassembly

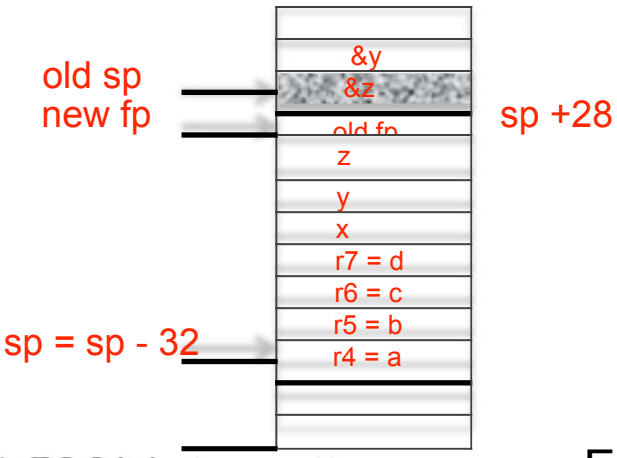
```
    int d, int *Pz, int *Py){
0x00040204 <Example1>:      addi sp,sp,-32
0x00040208 <Example1+4>:    stw  fp,28(sp)
0x0004020c <Example1+8>:    addi fp,sp,28
0x00040210 <Example1+12>:   stw  r4,-28(fp)
0x00040214 <Example1+16>:   stw  r5,-24(fp)
0x00040218 <Example1+20>:   stw  r6,-20(fp)
0x0004021c <Example1+24>:   stw  r7,-16(fp)
    int x, y, z;

    x = a + b;
0x00040220 <Example1+28>:   ldw  r3,-28(fp)
0x00040224 <Example1+32>:   ldw  r2,-24(fp)
0x00040228 <Example1+36>:   add  r2,r3,r2
0x0004022c <Example1+40>:   stw  r2,-12(fp)
    y = c - d;
0x00040230 <Example1+44>:   ldw  r3,-20(fp)
0x00040234 <Example1+48>:   ldw  r2,-16(fp)
```



Function Example1()

hello_world.c | MyFunction.c | myfunction.h

```
#include "myfunction.h"

void Example1(int a, int b, int c,
    int d, int *Pz, int *Py){
    int x, y, z;

    x = a + b;
    y = c - d;
    z = x * y;
    y = b + d;

    *Pz = z;
    *Py = y;
} /* Example1 */

int Example2(int a, int b, int c, int d){
    int x;

    if (a + b > 0) x = 5;
    else x = 7;

    return a * b + 5 * (c - d);
} /* Example2 */
```



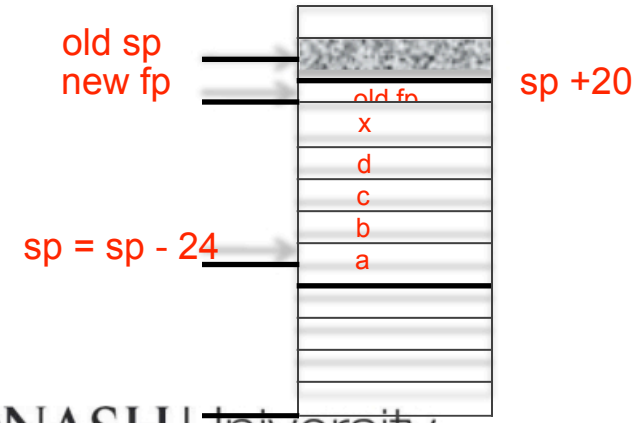
Outline | Make Targets | Disassembly

```
    int d, int *Pz, int *Py){
0x00040204 <Example1>:      addi sp,sp,-32
0x00040208 <Example1+4>:    stw  fp,28(sp)
0x0004020c <Example1+8>:    addi fp,sp,28
0x00040210 <Example1+12>:   stw  r4,-28(fp)
0x00040214 <Example1+16>:   stw  r5,-24(fp)
0x00040218 <Example1+20>:   stw  r6,-20(fp)
0x0004021c <Example1+24>:   stw  r7,-16(fp)
    int x, y, z;

    x = a + b;
0x00040220 <Example1+28>:   ldw  r3,-28(fp)
0x00040224 <Example1+32>:   ldw  r2,-24(fp)
0x00040228 <Example1+36>:   add  r2,r3,r2
0x0004022c <Example1+40>:   stw  r2,-12(fp)
    y = c - d;
0x00040230 <Example1+44>:   ldw  r3,-20(fp)
0x00040234 <Example1+48>:   ldw  r2,-16(fp)
0x00040238 <Example1+52>:   sub  r2,r3,r2
0x0004023c <Example1+56>:   stw  r2,-8(fp)
    z = x * y;
0x00040240 <Example1+60>:   ldw  r3,-12(fp)
0x00040244 <Example1+64>:   ldw  r2,-8(fp)
0x00040248 <Example1+68>:   mul  r2,r3,r2
0x0004024c <Example1+72>:   stw  r2,-4(fp)
    y = b + d;
0x00040250 <Example1+76>:   ldw  r3,-24(fp)
0x00040254 <Example1+80>:   ldw  r2,-16(fp)
0x00040258 <Example1+84>:   add  r2,r3,r2
0x0004025c <Example1+88>:   stw  r2,-8(fp)

    *Pz = z;
0x00040260 <Example1+92>:   ldw  r3,4(fp)
0x00040264 <Example1+96>:   ldw  r2,-4(fp)
0x00040268 <Example1+100>:  stw  r2,0(r3)
    *Py = y;
0x0004026c <Example1+104>:  ldw  r3,8(fp)
0x00040270 <Example1+108>:  ldw  r2,-8(fp)
0x00040274 <Example1+112>:  stw  r2,0(r3)
} /* Example1 */
0x00040278 <Example1+116>:  ldw  fp,28(sp)
0x0004027c <Example1+120>:  addi sp,sp,32
0x00040280 <Example1+124>:  ret
```

Function Example2()

hello_world.c | MyFunction.c | myfunction.h | printf.c

```
#include "myfunction.h"

void Example1(int a, int b, int c,
    int d, int *Pz, int *Py){
    int x, y, z;

    x = a + b;
    y = c - d;
    z = x * y;
    y = b + d;

    *Pz = z;
    *Py = y;
} /* Example1 */

int Example2(int a, int b, int c, int d){
    int x;

    if (a + b > 0) x = 5;
    else x = 7;

    return a * b + 5 * (c - d);
} /* Example2 */
```



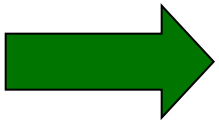
Outline | Make Targets | Disassembly

```
int Example2(int a, int b, int c, int d){
0x00040284 <Example2>:      addi   sp,sp,-24
0x00040288 <Example2+4>:    stw    fp,20(sp)
0x0004028c <Example2+8>:    addi   fp,sp,20
0x00040290 <Example2+12>:   stw    r4,-20(fp)
0x00040294 <Example2+16>:   stw    r5,-16(fp)
0x00040298 <Example2+20>:   stw    r6,-12(fp)
0x0004029c <Example2+24>:   stw    r7,-8(fp)
    int x;

    if (a + b > 0) x = 5;
0x000402a0 <Example2+28>:   ldw    r3,-20(fp)
0x000402a4 <Example2+32>:   ldw    r2,-16(fp)
0x000402a8 <Example2+36>:   add    r2,r3,r2
0x000402ac <Example2+40>:   cmplti r2,r2,1
0x000402b0 <Example2+44>:   bne    r2,zero,0x402c0 <Example2+60>
0x000402b4 <Example2+48>:   movi   r2,5
0x000402b8 <Example2+52>:   stw    r2,-4(fp)
0x000402bc <Example2+56>:   br     0x402c8 <Example2+68>
    else x = 7;
0x000402c0 <Example2+60>:   movi   r2,7
0x000402c4 <Example2+64>:   stw    r2,-4(fp)

    return a * b + 5 * (c - d);
0x000402c8 <Example2+68>:   ldw    r3,-20(fp)
0x000402cc <Example2+72>:   ldw    r2,-16(fp)
0x000402d0 <Example2+76>:   mul    r3,r3,r2
0x000402d4 <Example2+80>:   ldw    r4,-12(fp)
0x000402d8 <Example2+84>:   ldw    r2,-8(fp)
0x000402dc <Example2+88>:   sub    r2,r4,r2
0x000402e0 <Example2+92>:   muli   r2,r2,5
0x000402e4 <Example2+96>:   add    r3,r3,r2
0x000402e8 <Example2+100>:  mov    r2,r3
} /* Example2 */
0x000402ec <Example2+104>:  ldw    fp,20(sp)
0x000402f0 <Example2+108>:  addi   sp,sp,24
0x000402f4 <Example2+112>:  ret
```

Loop Unrolling

- **Goals:**
  - Reduce loop overhead (especially pipeline flushes)
  - Increase opportunities parallelism
  - Improve memory system performance

```
for(i=0; i<4; i++)
  {
  a[i] = b[i] * c[i];
  }
```



```
for(i=0; i<2; i++)
    {
    a[i*2] = b[i*2] * c[i*2];
    a[i*2+1] = b[i*2+1] * c[i*2+1];
    }
```

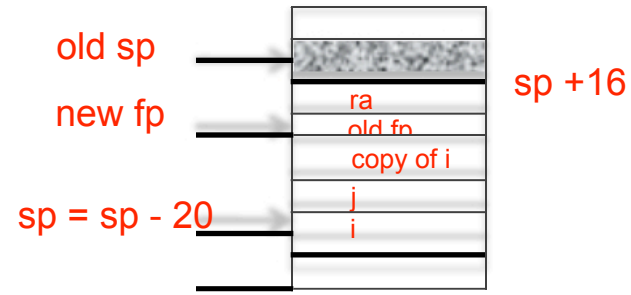
4 Branch instructions 2 Branch instructions

- **Downside is increased code size**

Compilers can produce inefficient code if not optimised.

NIOS code for a switch statement

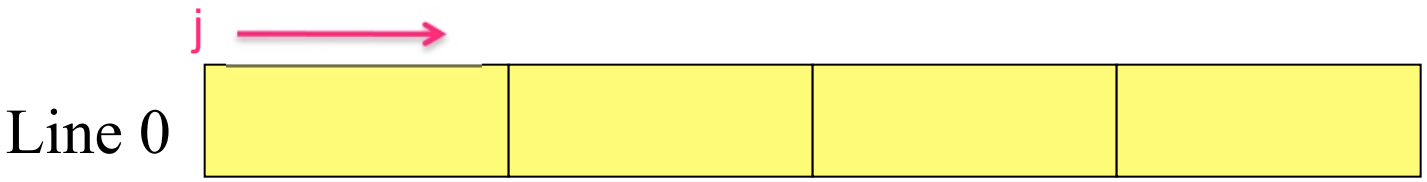
```
0x000091bc  addi    sp, sp, -0x14
0x000091c0  stw     ra, 16(sp)
0x000091c4  stw     fp, 12(sp)
0x000091c8  addi    fp, sp, 0xc
            int i,j;
            i=2;
0x000091cc  addi    r2, zero, 0x2
0x000091d0  stw     r2, -12(fp)
            switch(i){
0x000091d4  ldw     r2, -12(fp)
0x000091d8  stw     r2, -4(fp)
0x000091dc  ldw     r3, -4(fp)
0x000091e0  cmpeqi  r2, r3, 0x1
0x000091e4  bne     r2, zero, 0x38  (0x00009220)  // branch if 1
0x000091e8  ldw     r3, -4(fp)
0x000091ec  cmpgei  r2, r3, 0x2
0x000091f0  bne     r2, zero, 0x10  (0x00009204) // branch if >=2
0x000091f4  ldw     r3, -4(fp)
0x000091f8  cmpeq   r2, r3, zero
0x000091fc  bne     r2, zero, 0x14  (0x00009214)  // branch if 0
0x00009200  br      0x2c  (0x00009230)
0x00009204  ldw     r3, -4(fp)
0x00009208  cmpeqi  r2, r3, 0x2
0x0000920c  bne     r2, zero, 0x18  0x00009228) // branch if ==2
0x00009210  br      0x1c  (0x00009230)
            case 0:  j=3;
0x00009214  addi    r2, zero, 0x3
0x00009218  stw     -8(fp)
            break;
0x0000921c  br      0x10  (0x00009230)
            case 1:  j=0;
0x00009220  stw     zero, -8(fp)
            break;
0x00009224  br      0x8  (0x00009230)
            case 2:  j=1;
0x00009228  addi    r2, zero, -0x1
0x0000922c  stw     r2, -8(fp)
```



Cache reminder

- **A[8] accessed this way:**

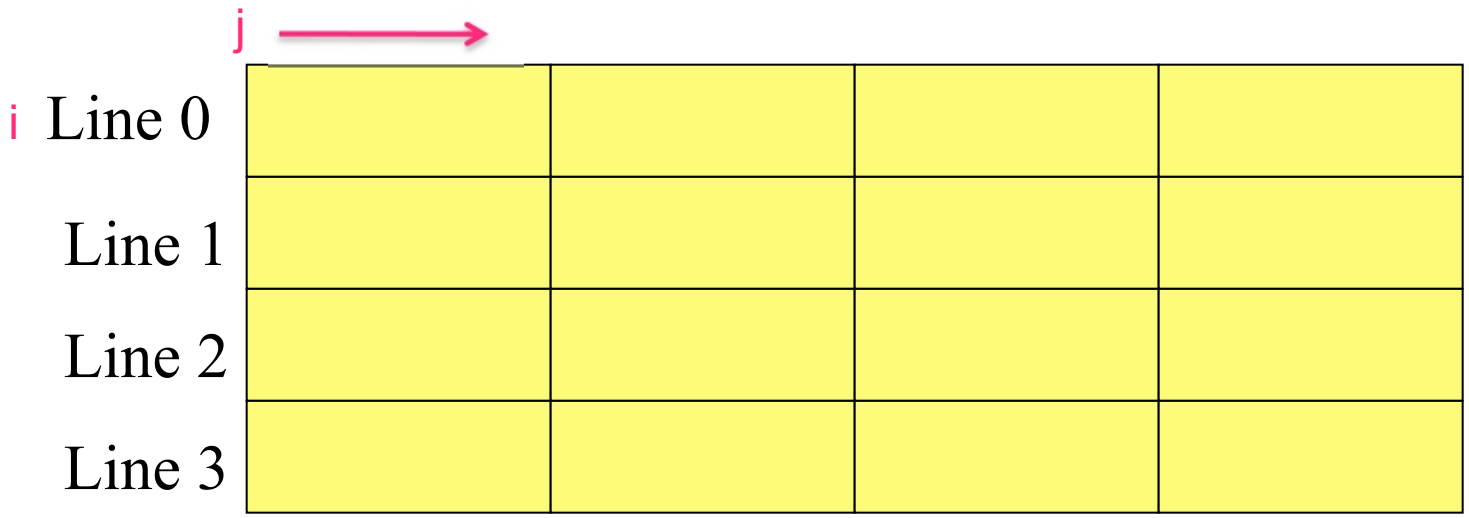
```
for(j = 0; j < 8; j++)
    A[j] += 20 ;
```



Array - Cache Problem (1/3)

- **A[8,8] accessed this way:**

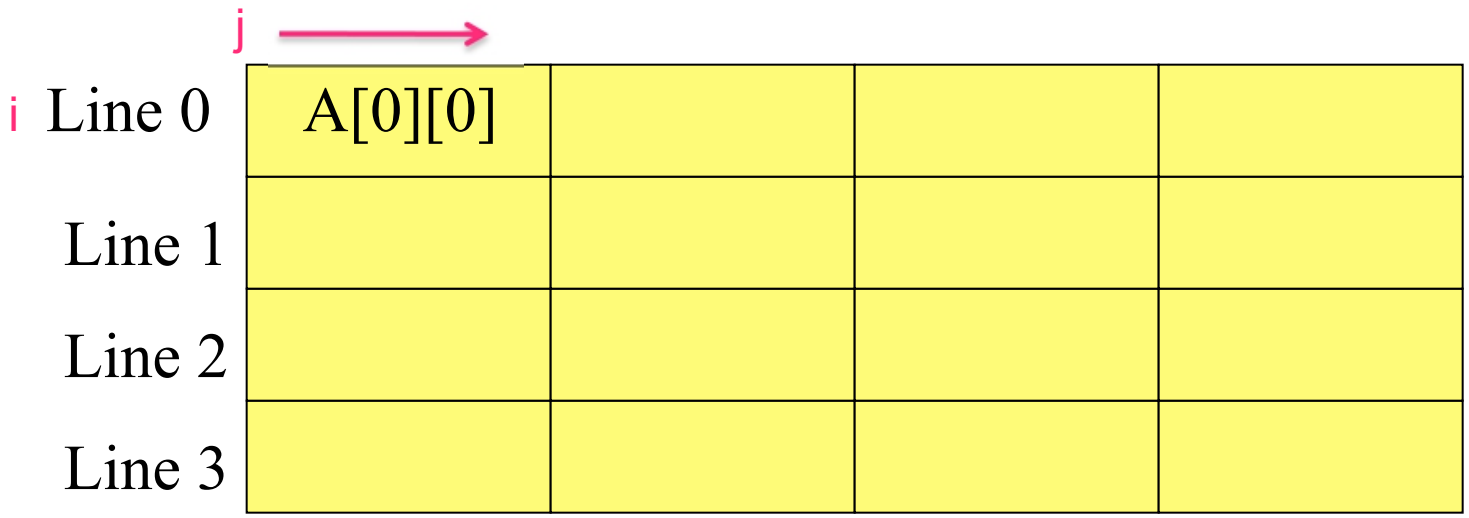
```
for(j = 0; j < 8; j++)
   for( i = 0; i < 8; i++)
      A[i][j] += 20 ;
```



Array - Cache Problem (1/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
   for( i = 0; i < 8; i++)
      A[i][j] += 20;
```



Array - Cache Problem (1/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
   for( i = 0; i < 8; i++)
      A[i][j] += 20 ;
```

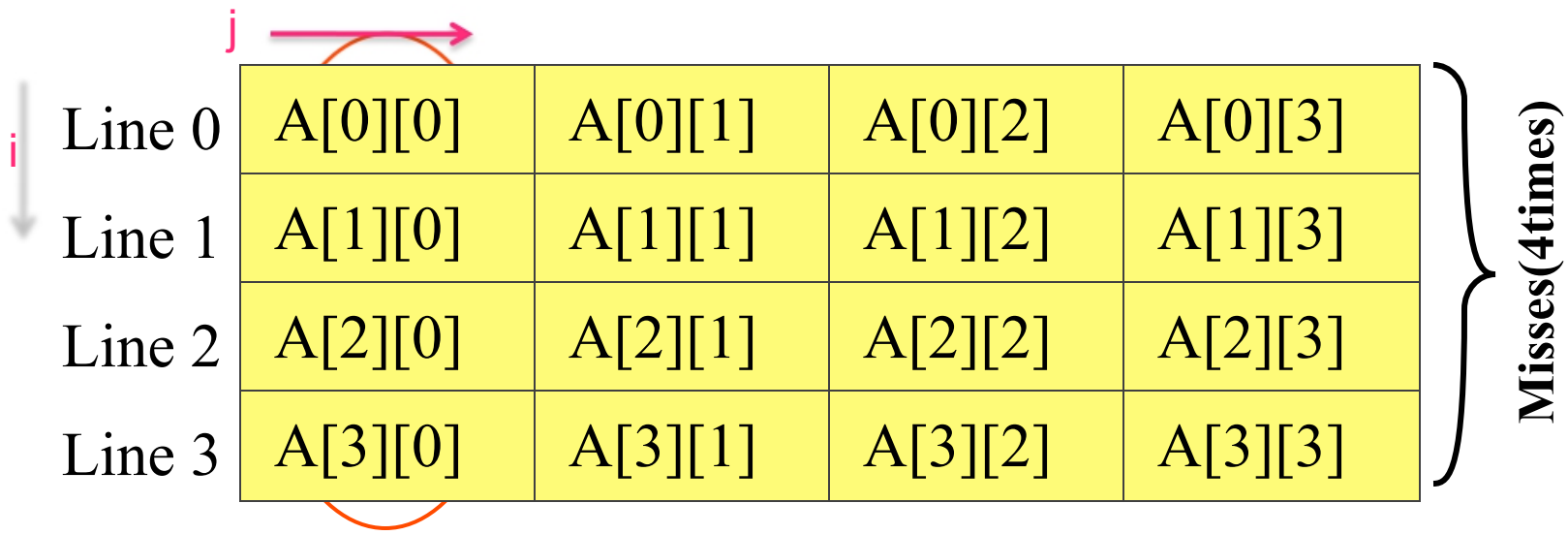
Array - Cache Problem (1/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
    for(i = 0; i < 8; i++)
        A[i][j] += 20;
```

Assume that the cache can hold 4 different entries and for each one includes the following 3 entries

- **After i = 3, j = 0 (4x i loop, 1x j loop)**



Array - Cache Problem (2/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
    for(i = 0; i < 8; i++)
        A[i][j] += 20;
```

Overwriting values introduced previously

- **After i = 7, j = 0 (8x i loop, 1x j loop)**

Line 4	A[4][0]	A[4][1]	A[4][2]	A[4][3]
Line 5	A[5][0]	A[5][1]	A[5][2]	A[5][3]
Line 6	A[6][0]	A[6][1]	A[6][2]	A[6][3]
Line 7	A[7][0]	A[7][1]	A[7][2]	A[7][3]

REPLACED!!(4times)

Array - Cache Problem (3/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
    for(i = 0; i < 8; i++)
        A[i][j] += 20;
```

Every time we cache a new value we also load 3 values we do not use

- **After i = 3, j = 1 (12x i loop, 2x j loop)**

Line 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Line 1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
Line 2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
Line 3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

REPLACED!!(4times)

Array - Cache Solution 1 (1/3)

- **A[8,8] accessed this way:**

```
for(i = 0; i < 8; i++)
    for(j = 0; j < 8; j++)
        A[i][j] += 20;
```

Swap i, j order

In this example order does not matter so we rearrange

- **After i = 3, j = 0 (4x i loop, 1x j loop)**

	miss	In cache	In cache	In cache
Line 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Line 1				
Line 2				
Line 3				

Miss (ONCE)

Array - Cache Solution 1 (2/3)

- **A[8,8] accessed this way:**

```
for(i = 0; i < 8; i++)
 for(j = 0; j < 8; j++)
    A[i][j] += 20;
```

Swap i, j order

- **After i = 7, j = 0 (8x i loop, 1x j loop)**

Line 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Line 1	A[0][4]	A[0][5]	A[0][6]	A[0][7]
Line 2				
Line 3				

Miss (ONCE)

Array - Cache Solution 1 (3/3)

- **A[8,8] accessed this way:**

```
for(j = 0; j < 8; j++)
   for(i = 0; i < 8; i++)
      A[i][j] += 20;
```

- **4 misses, 60 REPLACEMENTS!!**

- **Change order of access:**

```
for(i = 0; i < 8; i++)
 for(j = 0; j < 8; j++)
    A[i][j] += 20;
```

Swap i, j order

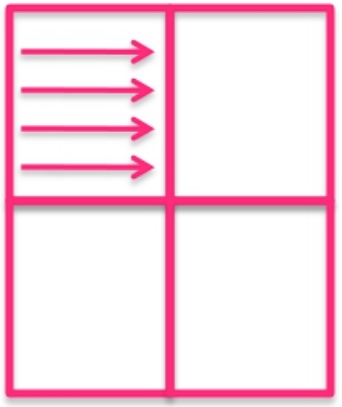
- **4 misses, 12 replacements, 48 HITS!!**

Array - Cache Solution 2

- **Loop tiling: break loop into a nest of loops**
- **A[8,8] accessed this way:**

```
for(q = 0; q < 2; q++)
  for(p = 0; p < 2; p++)
    for(j = (q * 4); j < ((q * 4) + 4); j++)
      for(i = (p * 4); i < ((p * 4) + 4); i++)
        A[i][j] += 20;
```

  - 4 misses, 12 replacements, 48 HITS!!



Order of access linked to cache organisation

- **Less efficient than changing loop order**
- **Feasible method for arrays bigger than cache and/or complex arrays**
  - Example: X[i][j] = A[i][k] + B[j][k] + C[i][j]

Interpreters and JIT Compilers

- **Interpreter : translates and executes program statements on - the - fly**
  - Visual Basic 6.0

  Java intermediate translation
- **JIT compiler : compiles small sections of code into instructions during program execution.**
  - Eliminates some translation overhead
  - Often requires more memory
  - Java